COMSC-210 Lecture Topic 1
Structs and Classes

Reference
Childs Ch. 1

`struct` vs `class`
same except public/private default
`struct` all members public by default
  leave it that way, by convention
`class` all members private by default
  but mix public and private members
BOTH are used to create "objects"

`struct` Data Members
for storing *multiple* values in a *single* variable
requires programmer to list the values
  name
  data type

```
struct CarType
{
  string maker;
  int year;
  float price;
};
```

NOTE: use of "string" requires `#include <string>`
above example has 3 "data members"
data members can even be:
  arrays
  pointers
  other `struct` variables

`struct` Member Functions
vs "stand-alone functions" -- e.g.:

```
struct CarType
{
  string maker;
  int year;
  float price;
  void output() const; // prototype
};

void CarType::output() const
{
  cout << "Make: " << maker...
}
```

about "inline" functions...
  do NOT use the `inline` keyword in COMSC 210
instead, write SHORT functions inline like this:

```
struct CarType
{
  ...
  void output() const {cout << "Make: " << maker...}
};
```

Using `struct` Variables, or "Objects"
variable declaration statements:
  `int x;` // an int variable
  `CarType myCar;` // a CarType object
using data members:
  `myCar.maker = "Ford";` // setting
  `cout << myCar.year;` // getting
  `myCar.output();` // using a member function
object assignment statements:
  `myNewCar = yourOldCar;` // copies each data member

Using Objects in Functions

A Specification File: Checkbook.h
note: `const` keyword for "getter" member functions!

```
#ifndef Checkbook_h
#define Checkbook_h

class Checkbook
{
  public:
  void setBalance(float amt); // a setter
  bool writeCheck(float amt); // a setter
  void deposit(float amt); // a setter
  float getBalance() const; // a getter
  float getLastCheck() const; // a getter
  float getLastDeposit() const; // a getter

  private:
  float balance; // dollars
  float lastCheck; // dollars
  float lastDeposit; // dollars
};

#endif
```

An Implemention File: Checkbook.cpp

```
#include "Checkbook.h"

void Checkbook::setBalance(float amt)
{
  balance = amt;
}

float Checkbook::getBalance() const
{
  return balance;
}
...
```

note these:
  `#include`, `Checkbook::`, `const`
  the variable `balance`
undeclared variables in function definitions
  are *data members* of the class

CheckbookDriver.cpp
from the textbook -- this is *not* a test driver
  so don't be misled by the file name

```
#include <iostream> // for ios and cout
#include <iomanip> // for setprecision
using namespace std;

#include "Checkbook.h"

int main()
{
  Checkbook cb;
  float newBalance;
  ...
  cb.setBalance(newBalance);
  ...
  cout.setf(ios::fixed|ios::showpoint);
  cout << setprecision(2) << cb.getBalance();
  ...
}
```

but do NOT use `cb.balance` directly

Writing a Test Driver
Follow this structure for your test drivers

```
// C++ library includes go here
#include <iostream>
...
```

as a parameter:
`void fun(CarType car)` // makes a mutable copy
`void fun(CarType& car)` // alias for original (mutable reference)
`void fun(const CarType& car)` // alias for original (read-only reference)
as a return value (stand-alone, value-returning function):

```
CarType getMyCar()
{
  CarType car;
  car.maker = "Ford";
  car.year = 1965;
  car.price = 1300;
  return car;
};
```

Beware of Gotchas!

`CarType& getMyCar()` returns a *reference*
never return reference to a local variable!
objects cannot be outputted
`cout << myCar;` // WILL NOT COMPILE
do NOT use scope resolution inside class definitions

```
class X
{
  ...
  void X::fun(); // NO X::
};
```

Implementing a `struct` or `class`

a class is a `struct` with private and public members
by convention, both are implemented in 2 files:
specification file (H)
implementation file (CPP)
...and then used in a 3rd file
by convention, avoid user I/O in structs and classes
do NOT rely on #includes from other files

When to Use a `#include` Statement

include the "string" library in any H or CPP that uses "string"
include the "cmath" library in any H or CPP that uses "sqrt", "pow", etc
include the "cstdlib" library in any H or CPP that uses "abs", "atoi", etc
do NOT rely on "dependencies" among included files
do NOT rely on whatever your compiler allows

```
using namespace std;

// C library includes go here
#include <cassert>
...

#include "MyClass.h" // class being tested
#include "MyClass.h" // ifndef test

int main()
{
  MyClass a;
  ...

  // object copy testing
  {
    const MyClass copy = a; // a read-only copy
    ...confirm that copy's contents match a's
  }

  // object assignment testing
  {
    MyClass copy; copy = a;
    ...confirm that copy's contents match a's
  }
}
```

Test only ONE class per driver CPP

Output actual and expected values
and use assertion to confirm results

```
// testing the fun function
cout << "Testing the fun function. Expected result is 100\n";
cout << "  Actual result is " << a.fun() << endl;
assert(100 == a.fun()); // halt program upon 1st mismatch
```

NOTE: use of "assert" requires `#include <cassert>`
Write test drivers for *all* your developed classes
Final version of test driver should test
final version of class
OK to have test `cout`s in class functions
but remove them in final version

Some Useful How-To's

See below -- use for future reference

ABOUT INLINE FUNCTIONS *show / hide*

How to debug using simple tracing

```
cout << __FILE__ << " at line " << __LINE__ << endl;
```

How to make sure that assertions are working

```
assert(false); // causes program to end here
```

How to format floating point output:

```
// set to 2 digits after the decimal
cout.setf(ios::fixed|ios::showpoint); // requires iostream
cout << setprecision(2); // requires iostream and iomanip

// unformat -- so show a number "as is"
cout.unsetf(ios::fixed|ios::showpoint); // requires iostream
cout << setprecision(6); // requires iostream and iomanip
```

How to output a value with leading zeros, justified in a 20-column space:

```
cout.setf(ios::left); // or ios::right or left/right "manipulators"
cout.fill('0'); // pad with zeros
cout << setw(20) << aValue;
```

How to access a C++ `string` as a C `char` array:

```
string cppString = ...; // the input (in the string library)
```

```
const char* cString = cppString.c_str(); // the C version of the C++ string
```

Note that `cString` is not mutable. If you want a mutable copy of the C++ `string`, add this code:

```
char cStringCopy[1000]; // hopefully cString is less than 1000 in length!
strcpy(cStringCopy, cString); // in cstring library
```

How to convert a space-delimited C++ `string` into an STL `queue` of string tokens:

```
const char* const space = " "; // the delimiter that separates tokens
string blankSeparatedText = "..."; // the input
queue<string> tokens; // the output
char buf[1000]; // a buffer for strtok to use -- assume no token longer than 999 chars
strcpy(buf, blankSeparatedText.c_str()); // copy string into char* for strtok to use, requires cstring library

// parse the text
char* token = strtok(buf, space); // first token, requires cstring library
while (token)
{
  tokens.push(string(token)); // copy char* to a string and add to queue
  token = strtok(0, space); // get token -- zero if no more
}
```

`strcpy` is in the `cstring` include, so is `strtok`. The directive has no effect on other compilers, so use it for cross-compiler compatibility. For it to be effective it must appear above *all* includes.

`queue` is in the `queue` include; `string` is in the `string` include.

How to convert a `string` of numeric format into an `int` or a `double`:

```
string sa = "101";
int ia = atoi(sa.c_str()); // requires cstdlib library

string sb = "3.14159";
double db = atof(sb.c_str()); // requires cstdlib library
```

`atoi` and `atof` are in the `cstdlib` include, and require *no* `using` statements.

How to convert an `int` or a `double` into a `string` using C formatting:

```
char buf[1000]; // a buffer -- assume no entry longer than 999 chars

int ia = 101;
sprintf(buf, "%d", ia); // in cstdio
string sa = buf; // this is for int's

double db = 3.14159;
sprintf(buf, "%f", db); // the buffer is reusable, requires cstdio library
string sb = buf; // this is for doubles
```

`sprintf` is in the `cstdio` include, and requires *no* `using` statement.

How to convert an `int` or a `double` into a `string` using C++ formatting:

```
stringstream sout; // requires sstream library
... // use sout as you would use cout
string sb = sout.str();
```

`stringstream` is in the `sstream` library.

Using Visual C++ 2010 For Win32 Console Applications